

llama.cpp KV Cache Investigation

Applying range-addressed KV cache allocation to the llama.cpp memory layer

Taeho Lee*
SSCCS Initiative
ssccs.org

August, 2026

1 Abstract

The llama.cpp investigation applies the tagma coordinate principle to the llama.cpp KV cache. The cache layer (`src/llama-kv-cache.*`) assigns each token a cache cell and passes per-token index vectors into the attention graph, the structural counterpart of the vLLM block table. A contiguous-slot mode already exists in the cache, which is the entry point for range-addressed allocation. The reference benchmark (llama-bench) runs on CPU and Apple Silicon without an accelerator, so the proof surface is GPU-free. This page records the measured Phase A and Phase B results and the insights derived from them.

2 Benchmark method

The host-path microbenchmark lives in the fork at `pocs/tagma-kv-bench` and drives the real `llama_kv_cache` functions (`find_slot`, `prepare`, `apply_ubatch`, `set_input_k_idx`s, `set_input_v_idx`s) with synthetic ubatches, so the attention arithmetic is excluded and the numbers are the host-side KV bookkeeping in isolation. The measurements run on the CPU-only build (GGML_METAL=OFF, Release) on Apple M1 Max with Qwen2.5-0.5B-Instruct Q4_K_M at 32K context, matching the llama-bench baseline samples. The raw output is `results/host-bench-ctx32768.txt` in the fork.

3 Measured results

3.1 Baseline samples

The llama-bench baseline samples on the CPU-only build, `-p <len> -n 256 -ngl 0`:

Context	Prefill tok/s (pp)	Decode tok/s (tg)
4096	648.7	157.5
32768	165.9	133.8

The 128K sample was dropped: on CPU it exceeds 45 minutes for a single sample, and the prefill drop is dominated by the quadratic attention cost, which is not a tagma signal.

*Founder & Architect; Corresponding author: lee@ssccs.org

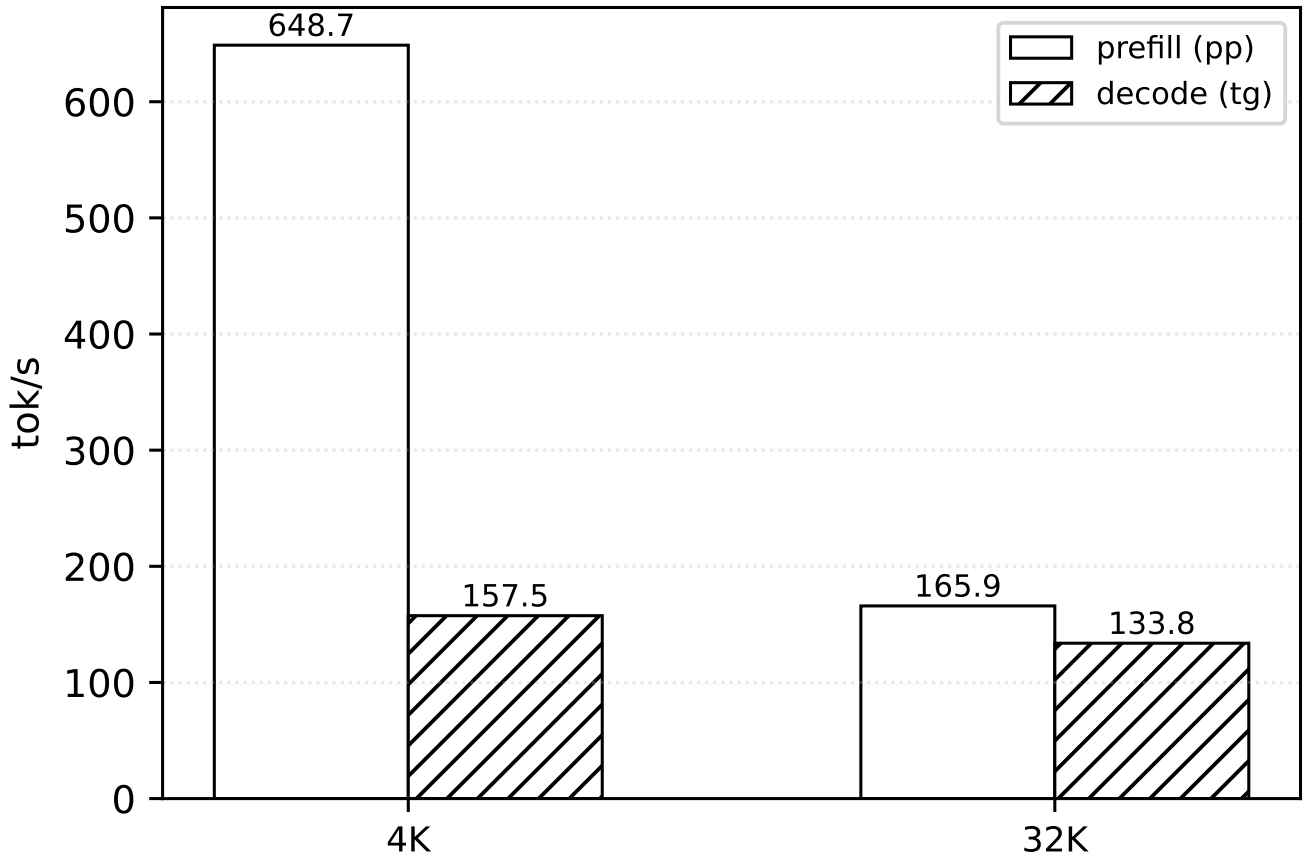


Figure 1: Baseline samples on the CPU-only build: prompt processing and decode throughput by context length. The prefill throughput drops by a factor of four from 4K to 32K, a drop dominated by the quadratic attention cost.

3.2 Host KV bookkeeping per decode step

State	find_slot(false)	find_slot(true)	prepare	apply_ubatch	set_input_k/v	host total	decode share
Single sequence, unified, clean ring	0.050 us	0.038 us	2.249 us	0.194 us	0.020/0.019 us	2.482 us	0.033 %
Single sequence, non-unified, clean ring	0.055 us	0.040 us	2.228 us	0.179 us	0.021/0.020 us	2.448 us	0.033 %
Fragmented, eviction with 32 survivors	0.935 us	192.7 us	17.06 us	2.558 us	0.061/0.026 us	19.70 us	0.26 %

The decode share references the 32K decode step of 7.47 ms (133.8 tok/s).

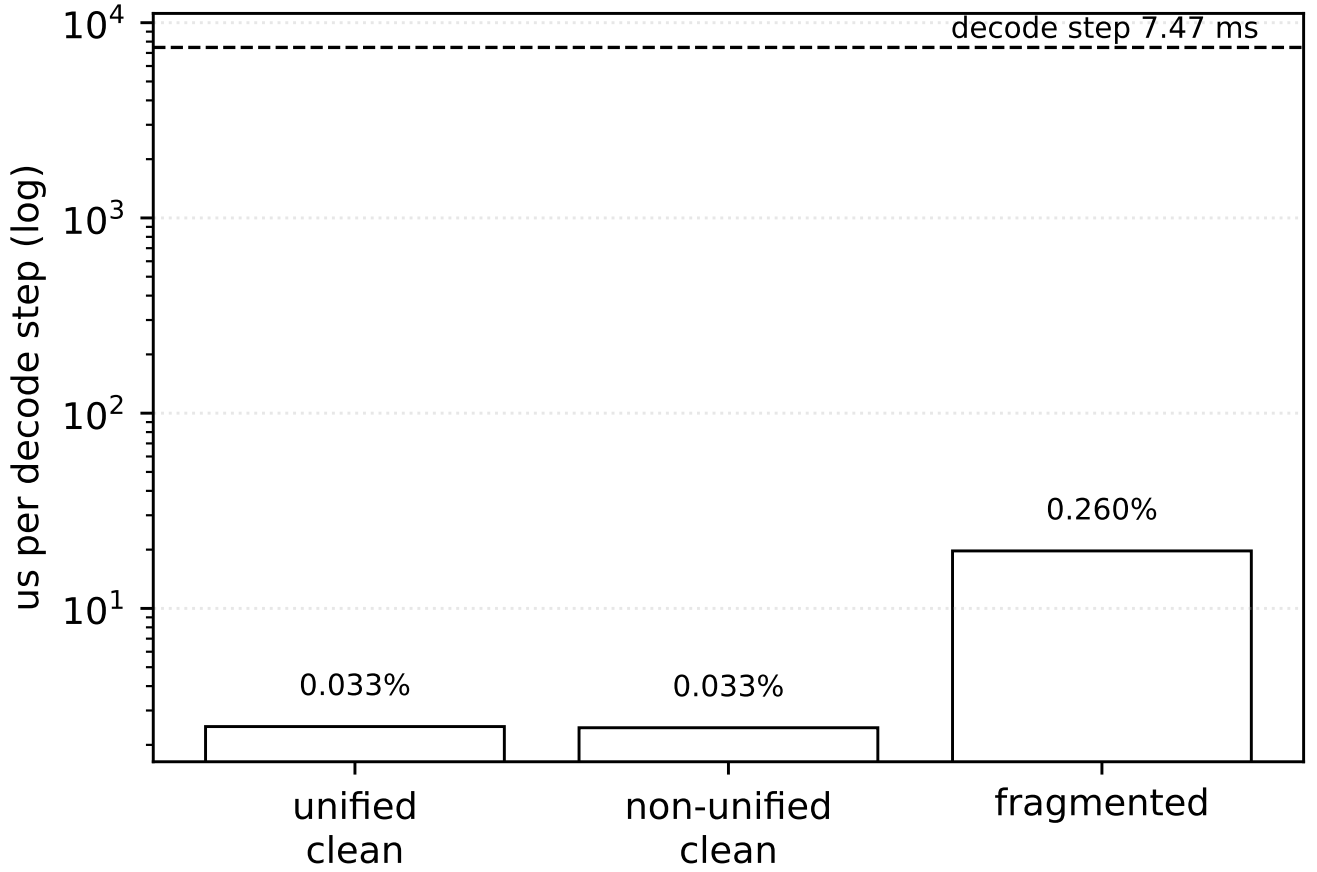


Figure 2: Host KV bookkeeping per decode step against the decode step itself, log scale. The three states span 2.4 to 19.7 microseconds; the 32K decode step is 7.47 milliseconds, three orders of magnitude above. The labels are the share of the decode step.

3.3 Contiguity outcomes under eviction

The eviction scenario fills the unified ring to 90% with 64 interleaved sequences, removes every other sequence, then decodes with the 32 survivors over 500 steps.

Metric	Value
Decode-slot contiguity, <code>is_contiguous</code>	8.0 %
<code>find_slot(cont=true)</code> success, a run of 32 cells exists	100 %
<code>find_slot(cont=true)</code> search cost	192.7 us
Per-sequence append adjacency, new cell equals previous cell plus one	0.0 %

3.4 Write scatter: `set_rows` versus dense copy

The comparison runs single-op graphs on the CPU backend with the thread count pinned to one, because `GGML_OP_SET_ROWS` always executes as a single task while `GGML_OP_CPY` executes with the backend thread count, and a four-thread single-op `CPY` graph pays a ~70 us thread-pool launch artifact that does not reflect the in-graph cost.

Layout	<code>set_rows</code>	dense copy	memcpy floor
K, 1 token	6.51 us	6.50 us	0.005 us

Layout	set_rows	dense copy	memcpy floor
K, 256 tokens	7.93 us	7.88 us	1.12 us
V transposed, 1 token (128 rows)	6.49 us	6.10 us	0.005 us
V transposed, 256 tokens (32768 rows)	80.6 us	7.95 us	1.05 us

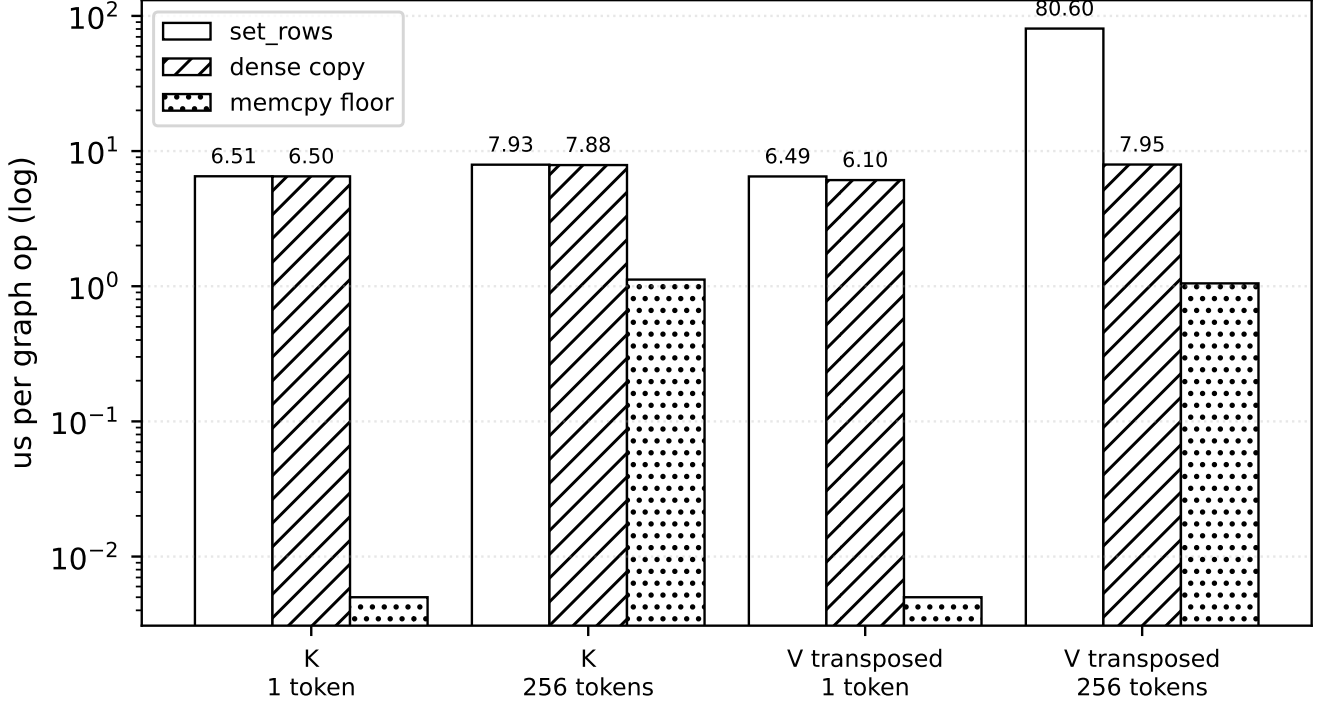


Figure 3: Write scatter cost at the ggml level, log scale: set_rows against a dense copy and the memcpy floor, per layout. The per-op graph dispatch dominates at about 6 microseconds for both scatter and dense copy at decode scale; only the transposed V layout at 256-token chunks shows a material gap.

3.5 In-graph scatter share of decode

The decode profile (`pocs/tagma-kv-bench/decode-prof.cpp`) runs a real 16K prompt prefill and 512 single-token decode steps, with env-gated timing of `ggml_compute_forward_set_rows` in the CPU backend (`TAGMA_PROF=1`).

Quantity	Value
set_rows calls (512 steps x 48 nodes)	24576
Average in-graph set_rows node time	0.700 us
set_rows per decode step	33.6 us
Measured share of decode wall (machine under load)	0.025 %
Share against the baseline decode step (7.47 ms at 32K)	0.45 %

The single-op graph benchmark overestimates the in-graph per-node cost by roughly an order of magnitude: the ~6 us fixed dispatch is amortized inside the real graph, and the node executes at ~0.7 us. The in-graph scatter share of decode is below one percent by either normalization.

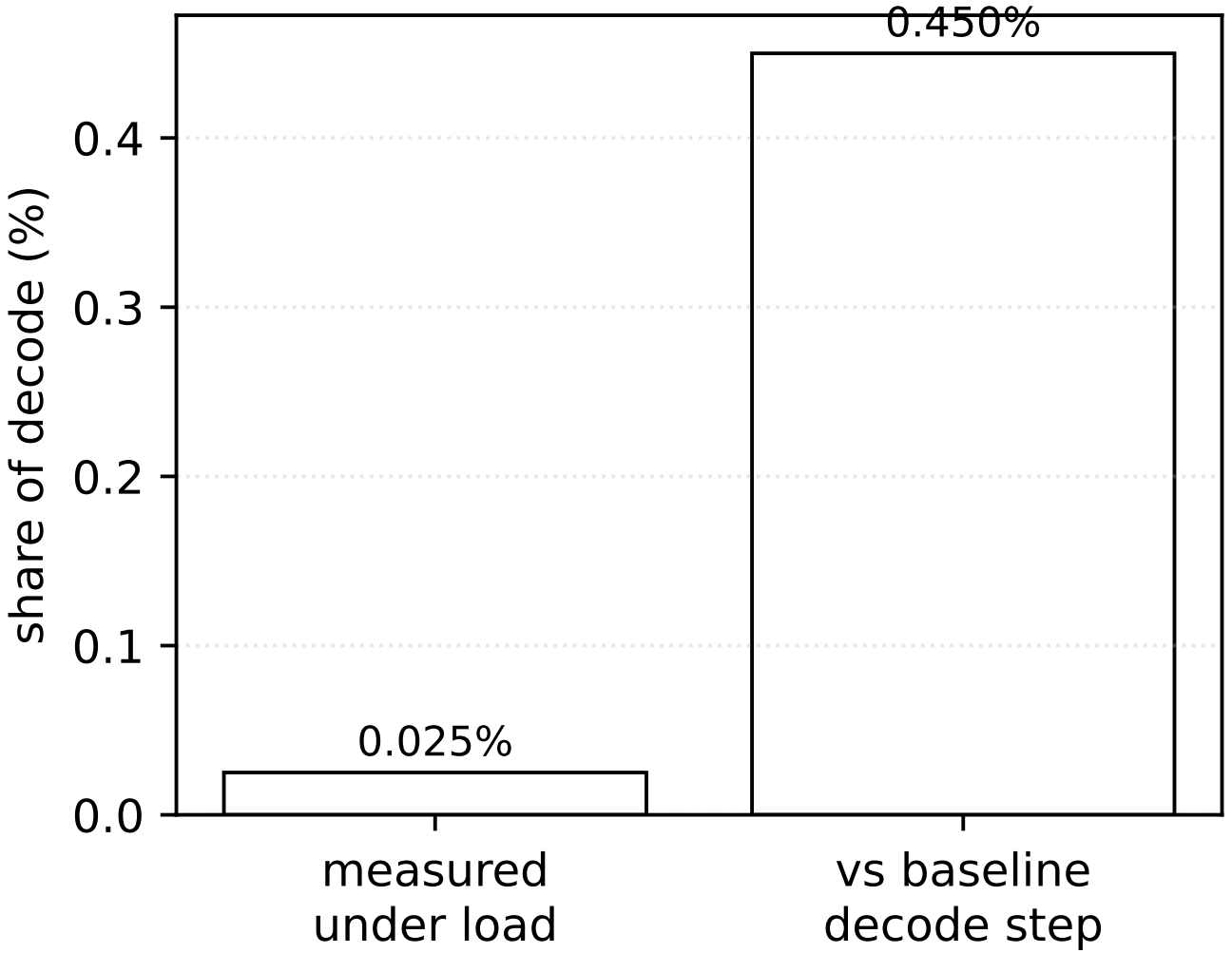


Figure 4: In-graph write scatter share of decode: 0.025 percent measured under load and 0.45 percent against the unloaded baseline decode step, both below one percent.

4 Insights

- The prepare snapshot and restore dominates the bookkeeping at 2.2 us of the 2.5 us per decode step and stays flat at ~0.5 us per token for chunked ubatches across fill factors from 25% to 97%.
- Fragmentation raises the base cell scan from 0.050 us to 0.935 us, a 19x increase, while the contiguous-run search rises from 0.038 us to 192.7 us. The contiguity verification is the expensive operation, and the default path never pays it.
- Unified mode interleaves the sequences, measured at 0.0% per-sequence adjacency and 8.0% decode-slot contiguity under eviction. The read path consumes the ring sequentially through the contiguous per-stream view, so the interleaving does not fragment the CPU read pattern.
- The per-op graph dispatch dominates the write scatter at ~6 us for both set_rows and the dense copy against a 0.005 us memcpy floor. The only material gap is the transposed V layout at 256-token chunks, 80.6 us versus 7.95 us, a prefill-only case at roughly 0.13% of the 32K prefill wall time.
- The llama.cpp KV path is array-indexed: cells, streams, and layers are dense arrays with arithmetic addressing. The hash-IO replacement that the tagma map work targets has no counterpart in this path, because the per-token indirection is a row scatter and no hash lookup participates in the KV path.

5 Status

The Phase A and Phase B measurements on the CPU-only build are recorded above. The Phase A gate asks whether the KV access path is a measurable share of decode time, and the measurements answer it: the host-side KV bookkeeping is 0.033 % of the decode step on a clean ring and 0.26 % under fragmentation, and the in-graph write scatter is 0.025 % measured and 0.45 % against the baseline step. The tagma range-addressed intervention removes exactly these costs, so within the llama.cpp KV path on the CPU surface it has no measurable performance basis. The read path is already range-based and the KV structures are array-indexed, so the hash-IO replacement that the tagma map work targets has no counterpart in this path.

The investigation continues. The remaining items are the SWA variant on a sliding-window model and the GPU-side behavior, which stays on the vLLM CUDA track; the hash-IO replacement value of the tagma map work is scoped as a separate investigation.

6 Conclusion

The Phase A gate measured a negative result on the tested CPU surface: the host-side KV bookkeeping and the write scatter sit at 0.025 to 0.45 percent of decode time, and the read path is already range-based. The tagma range-addressed intervention removes exactly these costs, so within the current llama.cpp KV path it has no measurable performance basis.

The entry point is structural and future. If llama.cpp adopts a vLLM-style paged KV cache, a shared block pool with per-sequence block tables, the block-table indirection that the current design lacks will appear, and the tagma range-arithmetic intervention gains a structural target: contiguous range allocation from the shared pool delivers the same memory efficiency without the block table. The measured lesson bounds the expectation for that scenario: the removable share equals the size of the indirection the paged design introduces, in the same 0.03 to 0.45 percent class measured here. The value framing is therefore paged-class memory efficiency without the block-table indirection, not a large speedup.

The adoption of a paged design is an upstream decision. The track remains parked with the open scope recorded in the fork investigation notes: the SWA variant, unified-mode locality, GPU-side behavior, and the paged-adoption trigger.